Nectar: Automatic Management of Data and Computation in Datacenters

Pradeep Kumar Gunda, Lenin Ravindranath*, Chandramohan A. Thekkath, Yuan Yu, Li Zhuang

Microsoft Research Silicon Valley

Abstract

Managing data and computation is at the heart of datacenter computing. Manual management of data can lead to data loss, wasteful consumption of storage, and laborious bookkeeping. Lack of proper management of computation can result in lost opportunities to share common computations across multiple jobs or to compute results incrementally.

Nectar is a system designed to address the aforementioned problems. It automates and unifies the management of data and computation within a datacenter. In Nectar, data and computation are treated interchangeably by associating data with its computation. Derived datasets, which are the results of computations, are uniquely identified by the programs that produce them, and together with their programs, are automatically managed by a datacenter wide caching service. Any derived dataset can be transparently regenerated by re-executing its program, and any computation can be transparently avoided by using previously cached results. This enables us to greatly improve datacenter management and resource utilization: obsolete or infrequently used derived datasets are automatically garbage collected, and shared common computations are computed only once and reused by others.

This paper describes the design and implementation of Nectar, and reports on our evaluation of the system using analytic studies of logs from several production clusters and an actual deployment on a 240-node cluster.

1 Introduction

Recent advances in distributed execution engines (MapReduce [7], Dryad [18], and Hadoop [12]) and high-level language support (Sawzall [25], Pig [24], BOOM [3], HIVE [17], SCOPE [6], DryadLINQ [29]) have greatly simplified the development of large-scale, data-intensive, distributed applications. However, major challenges still remain in realizing the full potential of data-intensive distributed computing within datacenters. In current practice, a large fraction of the computations in a datacenter is redundant and many datasets are obsolete or seldom used, wasting vast amounts of resources in a datacenter.

As one example, we quantified the wasted storage in our 240-node experimental Dryad/DryadLINQ cluster. We crawled this cluster and noted the last access time for each data file. We discovered that around 50% of the files was not accessed in the last 250 days.

As another example, we examined the execution statistics of 25 production clusters running data-parallel applications. We estimated that, on one such cluster, over 7000 hours of redundant computation can be eliminated per day by caching intermediate results. (This is approximately equivalent to shutting off 300 machines daily.) Cumulatively, over all clusters, this figure is over 35,000 hours per day.

Many of the resource issues in a datacenter arise due to lack of efficient management of either data or computation, or both. This paper describes Nectar: a system that manages the execution environment of a datacenter and is designed to address these problems.

A key feature of Nectar is that it treats data and computation in a datacenter interchangeably in the following sense. Data that has not been accessed for a long period may be removed from the datacenter and substituted by the computation that produced it. Should the data be needed in the future, the computation is rerun. Similarly, instead of executing a user's program, Nectar can partially or fully substitute the results of that computation with data already present in the datacenter. Nectar relies on certain properties of the programming environment in the datacenter to enable this interchange of data and computation.

Computations running on a Nectar-managed datacen-

*L. Ravindranath is affiliated with the Massachusetts Institute of Technology and was a summer intern on the Nectar project.

ter are specified as programs in LINQ [20]. LINQ comprises a set of operators to manipulate datasets of .NET objects. These operators are integrated into high level .NET programming languages (e.g., C#), giving programmers direct access to .NET libraries as well traditional language constructs such as loops, classes, and modules. The datasets manipulated by LINQ can contain objects of an arbitrary .NET type, making it easy to compute with complex data such as vectors, matrices, and images. All of these operators are *functional*: they transform input datasets to new output datasets. This property helps Nectar reason about programs to detect program and data dependencies. LINQ is a very expressive and flexible language, e.g., the MapReduce class of computations can be trivially expressed in LINQ.

Data stored in a Nectar-managed datacenter are divided into one of two classes: *primary* or *derived*. Primary datasets are created once and accessed many times. Derived datasets are the results produced by computations running on primary and other derived datasets. Examples of typical primary datasets in our datacenters are click and query logs. Examples of typical derived datasets are the results of thousands of computations performed on those click and query logs.

In a Nectar-managed datacenter, all access to a derived dataset is mediated by Nectar. At the lowest level of the system, a derived dataset is referenced by the LINQ program fragment or expression that produced it. Programmers refer to derived datasets with simple pathnames that contain a simple indirection (much like a UNIX symbolic link) to the actual LINQ programs that produce them. By maintaining this mapping between a derived dataset and the program that produced it, Nectar can reproduce any derived dataset after it is automatically deleted. Primary datasets are referenced by conventional pathnames, and are not automatically deleted.

A Nectar-managed datacenter offers the following advantages.

1. Efficient space utilization. Nectar implements a cache server that manages the storage, retrieval, and eviction of the results of all computations (i.e., derived datasets). As well, Nectar retains the description of the computation that produced a derived dataset. Since programmers do not directly manage datasets, Nectar has considerable latitude in optimizing space: it can remove unused or infrequently used derived datasets and recreate them on demand by rerunning the computation. This is a classic trade-off of storage and computation.

2. Reuse of shared sub-computations. Many applications running in the same datacenter share common sub-computations. Since Nectar automatically caches the results of sub-computations, they will be computed only once and reused by others. This significantly reduces redundant computations, resulting in better resource utilization.

3. Incremental computations. Many datacenter applications repeat the same computation on a sliding window of an incrementally augmented dataset. Again, caching in Nectar enables us to reuse the results of old data and only compute incrementally for the newly arriving data.

4. Ease of content management. With derived datasets uniquely named by LINQ expressions, and automatically managed by Nectar, there is little need for developers to manage their data manually. In particular, they do not have to be concerned about remembering the location of the data. Executing the LINQ expression that produced the data is sufficient to access the data, and incurs negligible overhead in almost all cases because of caching. This is a significant advantage because most datacenter applications consume a large amount of data from diverse locations and keeping track of the requisite filepath information is often a source of bugs.

Our experiments show that Nectar, on average, could improve space utilization by at least 50%. As well, incremental and sub-computations managed by Nectar provide an average speed up of 30% for the programs running on our clusters. We provide a detailed quantitative evaluation of the first three benefits in Section 4. We have not done a detailed user study to quantify the fourth benefit, but the experience from our initial deployment suggests that there is evidence to support the claim.

Some of the techniques we used such as dividing datasets into primary and derived and reusing the results of previous computations via caching are reminiscent of earlier work in version management systems [15], incremental database maintenance [5], and functional caching [16, 27]. Section 5 provides a more detailed analysis of our work in relation to prior research.

This paper makes the following contributions to the literature:

- We propose a novel and promising approach that automates and unifies the management of data and computation in a datacenter, leading to substantial improvements in datacenter resource utilization.

- We present the design and implementation of our system, including a sophisticated program rewriter and static program dependency analyzer.

- We present a systematic analysis of the performance of our system from a real deployment on 240-nodes as well as analytical measurements.

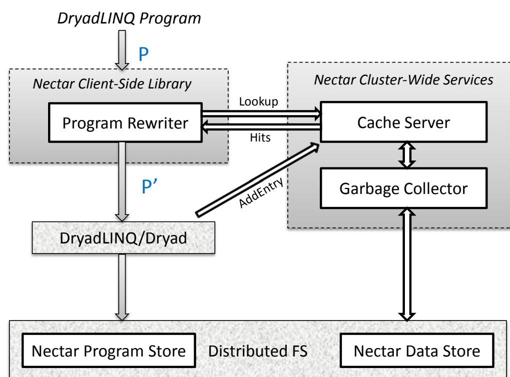

Figure 1: Nectar architecture. The system consists of a client-side library and cluster-wide services. Nectar relies on the services of DryadLINQ/Dryad and TidyFS, a distributed file system.

The rest of this paper is organized as follows. Section 2 provides a high-level overview of the Nectar system. Section 3 describes the implementation of the system. Section 4 evaluates the system using real workloads. Section 5 covers related work and Section 6 discusses future work and concludes the paper.

2 System Design Overview

The overall Nectar architecture is shown in Figure 1. Nectar consists of a client-side component that runs on the programmer's desktop, and two services running in the datacenter.

Nectar is completely transparent to user programs and works as follows. It takes a DryadLINQ program as input, and consults the cache service to rewrite it to an equivalent, more efficient program. Nectar then hands the resulting program to DryadLINQ which further compiles it into a Dryad computation running in the cluster. At run time, a Dryad job is a directed acyclic graph where vertices are programs and edges represent data channels. Vertices communicate with each other through data channels. The input and output of a DryadLINQ program are expected to be *streams*. A stream consists of an ordered sequence of extents, each storing a sequence of object of some data type. We use an in-house fault-tolerant, distributed file system called TidyFS to store streams.

Nectar makes certain assumptions about the underlying storage system. We require that streams be append-only, meaning that new contents are added by either appending to the last extent or adding a new extent. The metadata of a stream contains Rabin fingerprints [4] of the entire stream and its extents.

Nectar maintains and manages two namespaces in TidyFS. The program store keeps all DryadLINQ programs that have ever executed successfully. The data store is used to store all derived streams generated by DryadLINQ programs. The Nectar cache server provides cache hits to the program rewriter on the client side. It also implements a replacement policy that deletes cache entries of least value. Any stream in the data store that is not referenced by any cache entry is deemed to be garbage and deleted permanently by the Nectar garbage collector. Programs in the program store are never deleted and are used to recreate a deleted derived stream if it is needed in the future.

A simple example of a program is shown in Example 2.1. The program groups identical words in a large document into groups and applies an arbitrary user-defined function `Reduce` to each group. This is a typical MapReduce program. We will use it as a running example to describe the workings of Nectar. TidyFS, Dryad, and DryadLINQ are described in detail elsewhere [8, 18, 29]. We only discuss them briefly below to illustrate their relationships to our system.

In the example, we assume that the input D is a large (replicated) dataset partitioned as D_1, D_2 ... D_n in the TidyFS distributed file system and it consists of lines of text. `SelectMany` is a LINQ operator, which first produces a single list of output records for each input record and then "flattens" the lists of output records into a single list. In our example, the program applies the function `x => x.Split(' ')` to each line in D to produce the list of words in D.

The program then uses the `GroupBy` operator to group the words into a list of groups, putting the same words into a single group. `GroupBy` takes a *key-selector* function as the argument, which when applied to an input record returns a collating "key" for that record. `GroupBy` applies the key-selector function to each input record and collates the input into a list of groups (multisets), one group for all the records with the same key.

The last line of the program applies a transformation `Reduce` to each group. `Select` is a simpler version of `SelectMany`. Unlike the latter, `Select` produces a single output record (determined by the function `Reduce`) for each input record.

Example 2.1 A typical MapReduce job expressed in LINQ. `(x => x.Split(' '))` produces a list of blank-separated words; `(x => x)` produces a key for each input; `Reduce` is an arbitrary user supplied function that is applied to each input.

```
words = D.SelectMany(x => x.Split(' '));
groups = words.GroupBy(x => x);
result = groups.Select(x => Reduce(x));
```

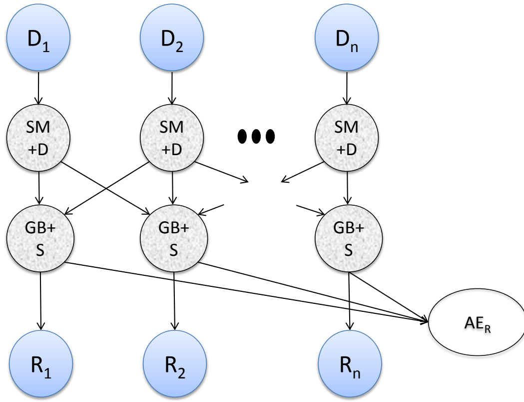

Figure 2: Execution graph produced by Nectar given the input LINQ program in Example 2.1. The nodes named `SM+D` executes `SelectMany` and distributes the results. `GB+S` executes `GroupBy` and `Select`.

When the program in Example 2.1 is run for the first time, Nectar, by invoking DryadLINQ, produces the distributed execution graph shown in Figure 2, which is then handed to Dryad for execution. (For simplicity of exposition, we assume for now that there are no cache hits when Nectar rewrites the program.) The `SM+D` vertex performs the `SelectMany` and distributes the results by partitioning them on a hash of each word. This ensures that identical words are destined to the same `GB+S` vertex in the graph. The `GB+S` vertex performs the `GroupBy` and `Select` operations together. The `AE` vertex adds a cache entry for the final result of the program. Notice that the derived stream created for the cache entry shares the same set of extents with the result of the computation. So, there is no additional cost of storage space. As a rule, Nectar always creates a cache entry for the final result of a computation.

2.1 Client-Side Library

On the client side, Nectar takes advantage of cached results from the cache to rewrite a program P to an equivalent, more efficient program P'. It automatically inserts *AddEntry* calls at appropriate places in the program so new cache entries can be created when P' is executed. The AddEntry calls are compiled into Dryad vertices that create new cache entries at runtime. We summarize the two main client-side components below.

Cache Key Calculation

A computation is uniquely identified by its program and inputs. We therefore use the Rabin fingerprint of the program and the input datasets as the cache key for a computation. The input datasets are stored in TidyFS and their fingerprints are calculated based on the actual stream contents. Nectar calculates the fingerprint of the program and combines it with the fingerprints of the input datasets to form the cache key.

The fingerprint of a DryadLINQ program must be able to detect any changes to the code the program depends on. However, the fingerprint should not change when code the program does not depend on changes. This is crucial for the correctness and practicality of Nectar. (Fingerprints can collide but the probability of a collision can be made vanishingly small by choosing long enough fingerprints.) We implement a static dependency analyzer to compute the transitive closure of all the code that can be reached from the program. The fingerprint is then formed using all reachable code. Of course, our analyzer only produces an over-approximation of the true dependency.

Rewriter

Nectar rewrites user programs to use cached results where possible. We might encounter different entries in the cache server with different sub-expressions and/or partial input datasets. So there are typically multiple alternatives to choose from in rewriting a DryadLINQ program. The rewriter uses a cost estimator to choose the best one from multiple alternatives (as discussed in Section 3.1).

Nectar supports the following two rewriting scenarios that arise very commonly in practice.

Common sub-expressions. Internally, a DryadLINQ program is represented as a LINQ expression tree. Nectar treats all prefix sub-expressions of the expression tree as candidates for caching and looks up in the cache for possible cache hits for every prefix sub-expression.

Incremental computations. Incremental computation on datasets is a common occurrence in data intensive computing. Typically, a user has run a program P on input D. Now, he is about to compute P on input $D + D'$, the concatenation of D and D'. The Nectar rewriter finds a new operator to combine the results of computing on the old input and the new input separately. See Section 2.3 for an example.

A special case of incremental computation that occurs in datacenters is a computation that executes on a sliding window of data. That is, the same program is repeatedly run on the following sequence of inputs:

$$Input_1 = d_1 + d_2 + ... + d_n,$$
$$Input_2 = d_2 + d_3 + ... + d_{n+1},$$
$$Input_3 = d_3 + d_4 + ... + d_{n+2},$$
$$......$$

Here d_i is a dataset that (potentially) consists of multiple extents distributed over many computers. So successive inputs to the program ($Input_i$) are datasets with some old extents removed from the head of the previous input and new extents appended to the tail of it. Nectar generates cache entries for each individual dataset d_i, and can use them in subsequent computations.

In the real world, a program may belong to a combination of the categories above. For example, an application that analyzes logs of the past seven days is rewritten as an incremental computation by Nectar, but Nectar may use sub-expression results of log preprocessing on each day from other applications.

2.2 Datacenter-Wide Service

The datacenter-wide service in Nectar comprises two separate components: the cache service and the garbage collection service. The actual datasets are stored in the distributed storage system and the datacenter-wide services manipulate the actual datasets by maintaining pointers to them.

Cache Service

Nectar implements a distributed datacenter-wide cache service for bookkeeping information about DryadLINQ programs and the location of their results. The cache service has two main functionalities: (1) serving the cache lookup requests by the Nectar rewriter; and (2) managing derived datasets by deleting the cache entries of least value.

Programs of all successful computations are uploaded to a dedicated program store in the cluster. Thus, the service has the necessary information about cached results, meaning that it has a recipe to recreate any derived dataset in the datacenter. When a derived dataset is deleted but needed in the future, Nectar recreates it using the program that produced it. If the inputs to that program have themselves been deleted, it backtracks recursively till it hits the immutable primary datasets or cached derived datasets. Because of this ability to recreate datasets, the cache server can make informed decisions to implement a cache replacement policy, keeping the cached results that yield the most hits and deleting the cached results of less value when storage space is low.

Garbage Collector

The Nectar garbage collector operates transparently to the users of the cluster. Its main job is to identify datasets unreachable from any cache entry and delete them. We use a standard mark-and-sweep collector. Actual content deletion is done in the background without interfering with the concurrent activities of the cache server and job executions. Section 3.2 has additional detail.

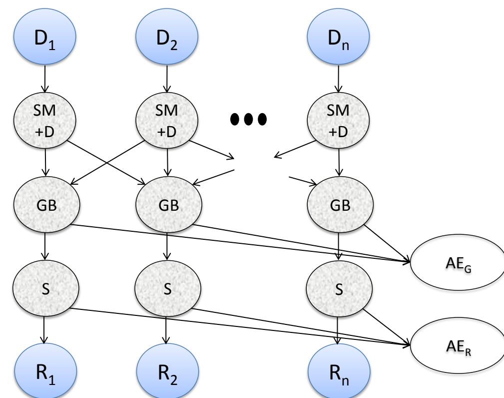

Figure 3: Execution graph produced by Nectar on the program in Example 2.1 after it elects to cache the results of computations. Notice that the `GroupBy` and `Select` are now encapsulated in separate nodes. The new `AE` vertex creates a cache entry for the output of `GroupBy`.

2.3 Example: Program Rewriting

Let us look at the interesting case of incremental computation by continuing Example 2.1.

After the program has been executed a sufficient number of times, Nectar may elect to cache results from some of its subcomputations based on the usage information returned to it from the cache service. So subsequent runs of the program may cause Nectar to create different execution graphs than those created previously for the same program. Figure 3 shows the new execution graph when Nectar chooses to cache the result of `GroupBy` (c.f. Figure 2). It breaks the pipeline of `GroupBy` and `Select` and creates an additional AddEntry vertex (denoted by `AE`) to cache the result of `GroupBy`. During the execution, when the `GB` stage completes, the `AE` vertex will run, creating a new TidyFS stream and a cache entry for the result of `GroupBy`. We denote the stream by G_D, partitioned as G_{D_1}, G_{D_2}, .. G_{D_n}.

Subsequently, assume the program in Example 2.1 is run on input $(D + X)$, where X is a new dataset partitioned as X_1, X_2,.. X_k. The Nectar rewriter would get a cache hit on G_D. So it only needs to perform `GroupBy` on X and merge with G_D to form new groups. Figure 4 shows the new execution graph created by Nectar.

There are some subtleties involved in the rewriting process. Nectar first determines that the number of partitions (n) of G_D. It then computes `GroupBy` on X the same way as G_D, generating n partitions with the same distribution scheme using the identical hash function as was used previously (see Figures 2 and 3). That is, the rewritten execution graph has k `SM+D` vertices, but n `GB`

vertices. The `MG` vertex then performs a pairwise merge of the output `GB` with the cached result G_D. The result of `MG` is again cached for future uses, because Nectar notices the pattern of incremental computation and expects that the same computation will happen on datasets of form G_{D+X+Y} in the future.

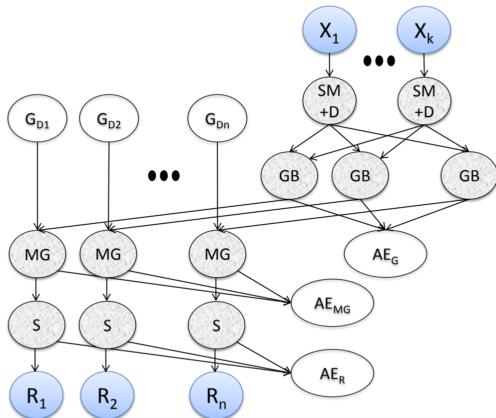

Figure 4: The execution graph produced by Nectar on the program in Example 2.1 on the dataset $D + X$. The dataset X consists of k partitions. The `MG` vertex merges groups with the same key. Both the results of `GB` and `MG` are cached. There are k `SM+D` vertices, but n `GB`, `MG`, and `S` vertices. $G_{D1}, ..., G_{Dn}$ are the partitions of the cached result.

Similar to MapReduce's combiner optimization [7] and Data Cube computation [10], DryadLINQ can decompose `Reduce` into the composition of two associative and commutative functions if `Reduce` is determined to be decomposable. We handle this by first applying the decomposition as in [28] and then the caching and rewriting as described above.

3 Implementation Details

We now present the implementation details of the two most important aspects of Nectar: Section 3.1 describes computation caching and Section 3.2 describes the automatic management of derived datasets.

3.1 Caching Computations

Nectar rewrites a DryadLINQ program to an equivalent but more efficient one using cached results. This generally involves: 1) identifying all sub-expressions of the expression, 2) probing the cache server for all cache hits for the sub-expressions, 3) using the cache hits to rewrite the expression into a set of equivalent expressions, and 4) choosing one that gives us the maximum benefit based on some cost estimation.

Cache and Programs

A cache entry records the result of executing a program on some given input. (Recall that a program may have more than one input depending on its arity.) The entry is of the form:

$$\langle FP_{PD}, FP_P, Result, Statistics, FPList \rangle$$

Here, FP_{PD} is the combined fingerprint of the program and its input datasets, FP_P is the fingerprint of the program only, $Result$ is the location of the output, and $Statistics$ contains execution and usage information of this cache entry. The last field $FPList$ contains a list of fingerprint pairs each representing the fingerprints of the first and last extents of an input dataset. We have one fingerprint pair for every input of the program. As we shall see later, it is used by the rewriter to search amongst cache hits efficiently. Since the same program could have been executed on different occasions on different inputs, there can be multiple cache entries with the same FP_P.

We use FP_{PD} as the primary key. So our caching is sound only if FP_{PD} can uniquely determine the result of the computation. The fingerprint of the inputs is based on the actual content of the datasets. The fingerprint of a dataset is formed by combining the fingerprints of its extents. For a large dataset, the fingerprints of its extents are efficiently computed in parallel by the datacenter computers.

The computation of the program fingerprint is tricky, as the program may contain user-defined functions that call into library code. We implemented a static dependency analyzer to capture all dependencies of an expression. At the time a DryadLINQ program is invoked, DryadLINQ knows all the dynamic linked libraries (DLLs) it depends on. We divide them into two categories: system and application. We assume system DLLs are available and identical on all cluster machines and therefore are not included in the dependency. For an application DLL that is written in native code (e.g., C or assembler), we include the entire DLL as a dependency. For soundness, we assume that there are no callbacks from native to managed code. For an application DLL that is in managed code (e.g., C#), our analyzer traverses the call graph to compute all the code reachable from the initial expression.

The analyzer works at the bytecode level. It uses standard .NET reflection to get the body of a method, finds all the possible methods that can be called in the body, and traverses those methods recursively. When a virtual method call is encountered, we include all the possible call sites. While our analysis is certainly a conservative approximation of the true dependency, it is reasonably

precise and works well in practice. Since dynamic code generation could introduce unsoundness into the analysis, it is forbidden in managed application DLLs, and is statically enforced by the analyzer.

The statistics information kept in the cache entry is used by the rewriter to find an *optimal* execution plan. It is also used to implement the cache insertion and eviction policy. It contains information such as the *cumulative execution time*, the number of hits on this entry, and the last access time. The cumulative execution time is defined as the sum of the execution time of all upstream Dryad vertices of the current execution stage. It is computed at the time of the cache entry insertion using the execution logs generated by Dryad.

The cache server supports a simple client interface. The important operations include: (1) `Lookup(fp)` finds and returns the cache entry that has `fp` as the primary key (FP_{PD}); (2) `Inquire(fp)` returns all cache entries that have `fp` as their FP_P; and (3) `AddEntry` inserts a new cache entry. We will see their uses in the following sections.

The Rewriting Algorithm

Having explained the structure and interface of the cache, let us now look at how Nectar rewrites a program.

For a given expression, we may get cache hits on any possible sub-expression and subset of the input dataset, and considering all of them in the rewriting is not tractable. We therefore only consider cache hits on prefix sub-expressions on segments of the input dataset. More concretely, consider a simple example `D.Where(P).Select(F)`. The `Where` operator applies a filter to the input dataset `D`, and the `Select` operator applies a transformation to each item in its input. We will only consider cache hits for the sub-expressions `S.Where(P)` and `S.Where(P).Select(F)` where `S` is a subsequence of extents in `D`.

Our rewriting algorithm is a simple recursive procedure. We start from the largest prefix sub-expression, the entire expression. Below is an outline of the algorithm. For simplicity of exposition, we assume that the expressions have only one input.

Step 1. For the current sub-expression E, we probe the cache server to obtain all the possible hits on it. There can be multiple hits on different subsequences of the input `D`. Let us denote the set of hits by H. Note that each hit also gives us its saving in terms of cumulative execution time. If there is a hit on the entire input `D`, we use that hit and terminate because it gives us the most savings in terms of cumulative execution time. Otherwise we execute Steps 2-4.

Step 2. We compute the best execution plan for E using hits on its smaller prefixes. To do that, we first compute the best execution plan for each immediate successor prefix of E by calling our procedure recursively, and then combine them to form a single plan for E. Let us denote this plan by (P_1, C_1) where C_1 is its saving in terms of cumulative execution time.

Step 3. For the H hits on E (from Step 1), we choose a subset of them such that (a) they operate on disjoint subsequence of `D`, and (b) they give us the most saving in terms of cumulative execution time. This boils down to the well-known problem of computing the maximum independent sets of an interval graph, which has a known efficient solution using dynamic programming techniques [9]. We use this subset to form another execution plan for E on `D`. Let us denote this plan by (P_2, C_2).

Step 4. The final execution plan is the one from P_1 and P_2 that gives us more saving.

In Step 1, the rewriter calls `Inquire` to compute H. As described before, `Inquire` returns all the possible cache hits of the program with different inputs. A useful hit means that its input dataset is identical to a subsequence of extents of `D`. A brute force search is inefficient and requires to check every subsequence. As an optimization, we store in the cache entry the fingerprints of the first and last extents of the input dataset. With that information, we can compute H in linear time.

Intuitively, in rewriting a program P on incremental data Nectar tries to derive a combining operator C such that $P(D+D') = C(P(D), D')$, where C combines the results of P on the datasets D and D'. Nectar supports all the LINQ operators DryadLINQ supports.

The combining functions for some LINQ operators require the parallel merging of multiple streams, and are not directly supported by DryadLINQ. We introduced three combining functions: `MergeSort`, `HashMergeGroups`, and `SortMergeGroups`, which are straightforward to implement using DryadLINQ's Apply operator [29]. `MergeSort` takes multiple sorted input streams, and merge sorts them. `HashMergeGroups` and `SortMergeGroups` take multiple input streams and merge groups of the same key from the input streams. If all the input streams are sorted, Nectar chooses to use `SortMergeGroups`, which is streaming and more efficient. Otherwise, Nectar uses `HashMergeGroups`. The `MG` vertex in Figure 4 is an example of this group merge.

The technique of reusing materialized views in database systems addresses a similar problem. One important difference is that a database typically does not maintain views for multiple versions of a table, which would prevent it from reusing results computed on old incarnations of the table. For example, suppose we have a materialized view V on D. When D is changed to $D + D_1$, the view is also updated to V'. So for any fu-

ture computation on $D + D_2$, V is no longer available for use. In contrast, Nectar maintains both V and V', and automatically tries to reuse them for any computation, in particular the ones on $D + D_2$.

Cache Insertion Policy

We consider every prefix sub-expression of an expression to be a candidate for caching. Adding a cache entry incurs additional cost if the entry is not useful. It requires us to store the result of the computation on disk (instead of possibly pipelining the result to the next stage), incurring the additional disk IO and space overhead. Obviously it is not practical to cache everything. Nectar implements a simple strategy to determine what to cache.

First of all, Nectar always creates a cache entry for the final result of a computation as we get it for free: it does not involve a break of the computation pipeline and incurs no extra IO and space overhead.

For sub-expression candidates, we wish to cache them only when they are predicted to be useful in the future. However, determining the potential usefulness of a cache entry is generally difficult. So we base our cache insertion policy on heuristics. The caching decision is made in the following two phases.

First, when the rewriter rewrites an expression, it decides on the places in the expression to insert AddEntry calls. This is done using the usage statistics maintained by the cache server. The cache server keeps statistics for a sub-expression based on request history from clients. In particular, it records the number of times it has been looked up. On response to a cache lookup, this number is included in the return value. We insert an AddEntry call for an expression only when the number of lookups on it exceeds a predefined threshold.

Second, the decision made by the rewriter may still be wrong because of the lack of information about the saving of the computation. Information such as execution time and disk consumption are only available at run time. So the final insertion decision is made based on the runtime information of the execution of the sub-expression. Currently, we use a simple benefit function that is proportional to the execution time and inversely proportional to storage overhead. We add the cache entry when the benefit exceeds a threshold.

We also make our cache insertion policy adaptive to storage space pressure. When there is no pressure, we choose to cache more aggressively as long as it saves machine time. This strategy could increase the useless cache entries in the cache. But it is not a problem because it is addressed by Nectar's garbage collection, discussed further below.

3.2 Managing Derived Data

Derived datasets can take up a significant amount of storage space in a datacenter, and a large portion of it could be unused or seldom used. Nectar keeps track of the usage statistics of all derived datasets and deletes the ones of the least value. Recall that Nectar permanently stores the program of every derived dataset so that a deleted derived can be recreated by re-running its program.

Data Store for Derived Data

As mentioned before, Nectar stores all derived datasets in a data store inside a distributed, fault-tolerant file system. The actual location of a derived dataset is completely opaque to programmers. Accessing an existing derived dataset must go through the cache server. We expose a standard file interface with one important restriction: New derived datasets can only be created as results of computations.

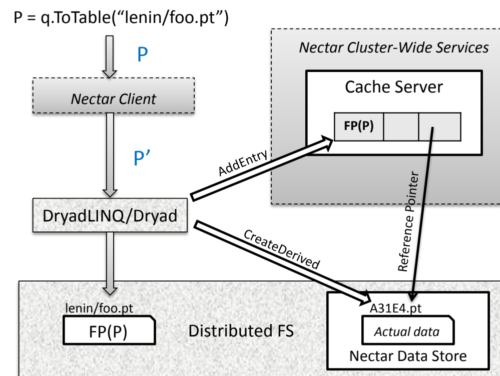

Figure 5: The creation of a derived dataset. The actual dataset is stored in the Nectar data store. The user file contains only the primary key of the cache entry associated with the derived.

Our scheme to achieve this is straightforward. Figure 5 shows the flow of creating a derived dataset by a computation and the relationship between the user file and the actual derived dataset. In the figure, `P` is a user program that writes its output to `lenin/foo.pt`. After applying transformations by Nectar and DryadLINQ, it is executed in the datacenter by Dryad. When the execution succeeds, the actual derived dataset is stored in the data store with a unique name generated by Nectar. A cache entry is created with the fingerprint of the program (`FP(P)`) as the primary key and the unique name as a field. The content of `lenin/foo.pt` just contains the primary key of the cache entry.

To access `lenin/foo.pt`, Nectar simply uses `FP(P)` to look up the cache to obtain the location of the actual derived dataset (`A31E4.pt`). The fact that all accesses go through the cache server allows us to keep

track of the usage history of every derived dataset and to implement automatic garbage collection for derived datasets based on their usage history.

Garbage Collection

When the available disk space falls below a threshold, the system automatically deletes derived datasets that are considered to be least useful in the future. This is achieved by a combination of the Nectar cache server and garbage collector.

A derived dataset is protected from garbage collection if it is referenced by any cache entry. So, the first step is to evict from the cache, entries that the cache server determines to have the least value.

The cache server uses information stored in the cache entries to do a cost-benefit analysis to determine the usefulness of the entries. For each cache entry, we keep track of the size of the resulting derived dataset (S), the elapsed time since it was last used (ΔT), the number of times (N) it has been used and the cumulative machine time (M) of the computation that created it. The cache server uses these values to compute the cost-to-benefit ratio

$$\text{CBRatio} = (S \times \Delta T)/(N \times M)$$

of each cache entry and deletes entries that have the largest ratios so that the cumulative space saving reaches a predefined threshold.

Freshly created cache entries do not contain information for us to compute a useful cost/benefit ratio. To give them a chance to demonstrate their usefulness, we exclude them from deletion by using a lease on each newly created cache entry.

The entire cache eviction operation is done in the background, concurrently with any other cache server operations. When the cache server completes its eviction, the garbage collector deletes all derived datasets not protected by a cache entry using a simple mark-and-sweep algorithm. Again, this is done in the background, concurrently with any other activities in the system.

Other operations can run concurrently with the garbage collector and create new cache entries and derived datasets. Derived datasets pointed to by cache entries (freshly created or otherwise) are not candidates for garbage collection. Notice however that freshly created derived datasets, which due to concurrency may not yet have a cache entry, also need to protected from garbage collection. We do this with a lease on the dataset.

With these leases in place, garbage collection is quite straightforward. We first compute the set of all derived datasets (ignoring the ones with unexpired leases) in our data store, exclude from it the set of all derived datasets referenced by cache entries, and treat the remaining as garbage.

Our system could mistakenly delete datasets that are subsequently requested, but these can be recreated by re-executing the appropriate program(s) from the program store. Programs are stored in binary form in the program store. A program is a complete Dryad job that can be submitted to the datacenter for execution. In particular, it includes the execution plan and all the application DLLs. We exclude all system DLLs, assuming that they are available on the datacenter machines. For a typical datacenter that runs 1000 jobs daily, our experience suggests it would take less than 1TB to store one year's program (excluding system DLLs) in uncompressed form. With compression, it should take up roughly a few hundreds of gigabytes of disk space, which is negligible even for a small datacenter.

4 Experimental Evaluation

We evaluate Nectar running on our 240-node research cluster as well as present analytic results of execution logs from 25 large production clusters that run jobs similar to those on our research cluster. We first present our analytic results.

4.1 Production Clusters

We use logs from 25 different clusters to evaluate the usefulness of Nectar. The logs consist of detailed execution statistics for 33182 jobs in these clusters for a recent 3-month period. For each job, the log has the source program and execution statistics such as computation time, bytes read and written and the actual time taken for every stage in a job. The log also gives information on the submission time, start time, end time, user information, and job status.

Programs from the production clusters work with massive datasets such as click logs and search logs. Programs are written in a language similar to DryadLINQ in that each program is a sequence of SQL-like queries [6]. A program is compiled into an expression tree with various stages and modeled as a DAG with vertices representing processes and edges representing data flows. The DAGs are executed on a Dryad cluster, just as in our Nectar managed cluster. Input data in these clusters is stored as append-only streams.

Benefits from Caching

We parse the execution logs to recreate a set of DAGs, one for each job. The root of the DAG represents the input to the job and a path through the DAG starting at the root represents a partial (i.e., a sub-) computation of the job. Identical DAGs from different jobs represent an opportunity to save part of the computation time of a later job by caching results from the earlier ones. We simulate

the effect of Nectar's caching on these DAGs to estimate cache hits.

Our results show that on average across all clusters, more than 35% of the jobs could benefit from caching. More than 30% of programs in 18 out of 25 clusters could have at least one cache hit, and there were even some clusters where 65% of programs could have cache hits.

The log contains detailed computation time information for each node in the DAG for a job. When there is a cache hit on a sub-computation of a job, we can therefore calculate the time saved by the cache hit. We show the result of this analysis in two different ways: Figure 6 shows the percentage of computing time saved and Table 1 shows the minimum number of hours of computation saved in each cluster.

Figure 6 shows that significant percentage of computation time can be saved in each cluster with Nectar. Most clusters can save a minimum of 20% to 40% of computation time and in some clusters the savings are up to 50%. Also, as an example, Table 1 shows a minimum of 7143 hours of computation per day can be saved using Nectar in Cluster C5. This is roughly equivalent to saying that about 300 machines in that cluster were doing wasteful computations all day that caching could eliminate. Across all 25 clusters, 35078 hours of computation per day can be saved, which is roughly equivalent to saving 1461 machines.

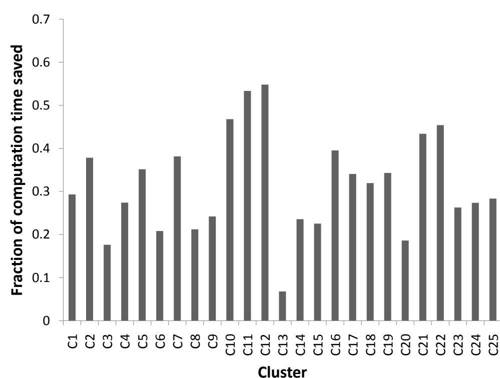

Figure 6: Fraction of compute time saved in each cluster

Ease of Program Development

Our analysis of the caching accounted for both sub-computation as well as incremental/sliding window hits. We noticed that the percentage of sliding window hits in some production clusters was minimal (under 5%). We investigated this further and noticed that many programmers explicitly structured their programs so that they can reuse a previous computation. This somewhat artificial structure makes their programs cumbersome, which can be alleviated by using Nectar.

Cluster	Computation Time Saved (hours/day)	Cluster	Computation Time Saved (hours/day)
C1	3898	C14	753
C2	2276	C15	755
C3	977	C16	2259
C4	1345	C17	3385
C5	7143	C18	528
C6	62	C19	4
C7	57	C20	415
C8	590	C21	606
C9	763	C22	2002
C10	2457	C23	1316
C11	1924	C24	291
C12	368	C25	58
C13	105		

Table 1: Minimum Computation Time Savings

There are anecdotes of system administrators manually running a common sub-expression on the daily input and explicitly notifying programmers to avoid each program performing the computation on its own and tying up cluster resources. Nectar automatically supports incremental computation and programmers do not need to code them explicitly. As discussed in Section 2, Nectar tries to produce the best possible query plan using the cached results, significantly reducing computation time, at the same time making it opaque to the user.

An unanticipated benefit of Nectar reported by our users on the research cluster was that it aids in debugging during program development. Programmers incrementally test and debug pieces of their code. With Nectar the debugging time significantly improved due to cache hits. We quantify the effect of this on the production clusters. We assumed that a program is a debugged version of another program if they had almost the same queries accessing the same source data and writing the same derived data, submitted by the same user and had the same program name.

Table 2 shows the amount of debugging time that can be saved by Nectar in the 90 day period. We present results for the first 12 clusters due to space constraints. Again, these are conservative estimates but shows substantial savings. For instance, in Cluster C1, a minimum of 3 hours of debugging time can be saved per day. Notice that this is actual elapsed time, i.e., each day 3 hours of computation on the cluster spent on debugging programs can be avoided with Nectar.

Cluster	Debugging Time Saved (hours)	Cluster	Debugging Time Saved (hours)
C1	270	C7	3
C2	211	C8	35
C3	24	C9	84
C4	101	C10	183
C5	94	C11	121
C6	8	C12	49

Table 2: Actual elapsed time saved on debugging in 90 days.

Managing Storage

Today, in datacenters, storage is manually managed.[1] We studied storage statistics in our 240-node research cluster that has been used by a significant number of users over the last 2 to 3 years. We crawled this cluster for derived objects and noted their last access times. Of the 109 TB of derived data, we discovered that about 50% (54.5 TB) was never accessed in the last 250 days. This shows that users often create derived datasets and after a point, forget about them, leaving them occupying unnecessary storage space.

We analyzed the production logs for the amount of derived datasets written. When calculating the storage occupied by these datasets, we assumed that if a new job writes to the same dataset as an old job, the dataset is overwritten. Figure 7 shows the growth of derived data storage in cluster C1. It show an approximately linear growth with the total storage occupied by datasets created in 90 days being 670 TB.

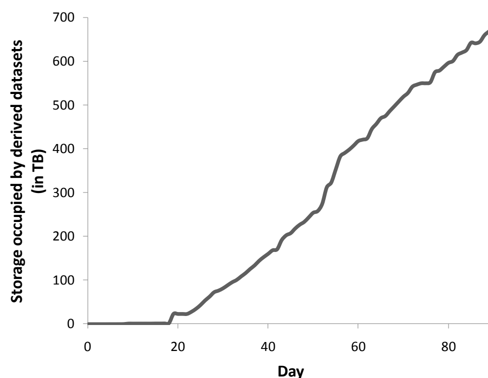

Figure 7: Growth of storage occupied by derived datasets in Cluster C1

[1]Nectar's motivation in automatically managing storage partly stems from the fact that we used to get periodic e-mail messages from the administrators of the production clusters requesting us to delete our derived objects to ease storage pressure in the cluster.

Cluster	Projected unreferenced derived data (in TB)
C1	2712
C5	368
C8	863
C13	995
C15	210

Table 3: Projected unreferenced data in 5 production clusters

Assuming similar trends in data access time in our local cluster and on the production clusters, Table 3 shows the projected space occupied by unreferenced derived datasets in 5 production clusters that showed a growth similar to cluster C1. Any object that has not been referenced in 250 days is deemed unreferenced. This result is obtained by extrapolating the amount of data written by jobs in 90 days to 2 years based on the storage growth curve and predicting that 50% of that storage will not be accessed in the last 250 days (based on the result from our local cluster). As we see, production clusters create a large amount of derived data, which if not properly managed can create significant storage pressure.

4.2 System Deployment Experience

Each machine in our 240-node research cluster has two dual-core 2.6GHz AMD Opteron 2218 HE CPUs, 16GB RAM, four 750GB SATA drives, and runs Windows Server 2003 operating system. We evaluate the comparative performance of several programs with Nectar turned on and off.

We use three datasets to evaluate the performance of Nectar:

WordDoc Dataset. The first dataset is a collection of Web documents. Each document contains a URL and its content (as a list of words). The data size is 987.4 GB . The dataset is randomly partitioned into 236 partitions. Each partition has two replicas in the distributed file system, evenly distributed on 240 machines.

ClickLog Dataset. The second dataset is a small sample from an anonymized click log of a commercial search engine collected over five consecutive days. The dataset is 160GB in size, randomly partitioned into 800 partitions, two replicas each, evenly distributed on 240 machines.

SkyServer Dataset. This database is taken from the Sloan Digital Sky Survey database [11]. It contains two data files: 11.8 and 41.8 GBytes of data. Both files were manually range-partitioned into 40 partitions using the same keys.

Sub-computation Evaluation

We have four programs: *WordAnalysis*, *TopWord*, *MostDoc*, and *TopRatio* that analyze the *WordDoc* dataset.

WordAnalysis parses the dataset to generate the number of occurrences of each word and the number of documents that it appears in. *TopWord* looks for the top ten most commonly used words in all documents. *MostDoc* looks for the top ten words appearing in the largest number of documents. *TopRatio* finds the percentage of occurrences of the top ten mostly used words among all words. All programs take the entire 987.4 GB dataset as input.

Program Name	Cumulative Time		Saving
	Nectar on	Nectar off	
TopWord	16.1m	21h44m	98.8%
MostDoc	17.5m	21h46m	98.6%
TopRatio	21.2m	43h30m	99.2%

Table 4: Saving by sharing a common sub-computation: Document analysis

With Nectar on, we can cache the results of executing the first program, which spends a huge amount of computation analyzing the list of documents to output an aggregated result of much smaller size (12.7 GB). The subsequent three programs share a sub-computation with the first program, which is satisfied from the cache. Table 4 shows the cumulative CPU time saved for the three programs. This behavior is not isolated, one of the programs that uses the *ClickLog* dataset shows a similar pattern; we do not report the results here for reasons of space.

Incremental Computation

We describe the performance of a program that studies query relevance by processing the *ClickLog* dataset. When users search a phrase at a search engine, they click the most relevant URLs returned in the search results. Monitoring the URLs that are clicked the most for each search phrase is important to understand query relevance. The input to the query relevance program is the set of all click logs collected so far, which increases each day, because a new log is appended daily to the dataset. This program is an example where the initial dataset is large, but the incremental updates are small.

Table 5 shows the cumulative CPU time with Nectar on and off, the size of datasets and incremental updates each day. We see that the total size of input data increases each day, while the computation resource used daily increases much slower when Nectar is on. We observed similar performance results for another program that calculates the number of active users, who are those that clicked at least one search result in the past three days. These results are not reported here for reasons of space.

	Data Size(GB)		Time (m)		Saving
	Total	Update	On	Off	
Day3	68.20	40.50	93.0	107.5	13.49%
Day4	111.25	43.05	112.9	194.0	41.80%
Day5	152.19	40.94	164.6	325.8	49.66%

Table 5: Cumulative machine time savings for incremental computation.

Debugging Experience: Sky Server

Here we demonstrate how Nectar saves program development time by shortening the debugging cycle. We select the most time-consuming query (Q18) from the Sloan Digital Sky Survey database [11]. The query identifies a gravitational lens effect by comparing the locations and colors of stars in a large astronomical table, using a three-way Join over two input tables containing 11.8 GBytes and 41.8 GBytes of data, respectively. The query is composed of four steps, each of which is debugged separately. When debugging the query, the first step failed and the programmer modified the code. Within a couple of tries, the first step succeeded, and execution continued to the second step, which failed, and so on.

Table 6 shows the average savings in cumulative time as each step is successively debugged with Nectar. Towards the end of the program, Nectar saves as much 88% of the time.

	Cumulative Time		Saving
	Nectar on	Nectar off	
Step 1	47.4m	47.4m	0%
Steps 1–2	26.5m	62.5m	58%
Steps 1–3	35.5m	122.7m	71%
Steps 1–4	15.0m	129.3m	88%

Table 6: Debugging: SkyServer cumulative time

5 Related Work

Our overall system architecture is inspired by the Vesta system [15]. Many high-level concepts and techniques (e.g., the notion of primary and derived data) are directly taken from Vesta. However, because of the difference in application domains, the actual design and implementation of the main system components such as caching and program rewriting are radically different.

Many aspects of query rewriting and caching in our work are closely related to incremental view maintenance and materialized views in the database literature [2, 5, 13, 19]. However, there are some important differences as discussed in Section 3.1. Also, we are not aware of the implementation of these ideas in systems

at the scale we describe in this paper. Incremental view maintenance is concerned with the problem of updating the materialized views incrementally (and consistently) when data base tables are subjected to random updates. Nectar is simpler in that we only consider append-only updates. On the other hand, Nectar is more challenging because we must deal with user-defined functions written in a general-purpose programming language. Many of the sophisticated view reuses given in [13] require analysis of the SQL expressions that is difficult to do in the presence of user-defined functions, which are common in our environment.

With the wide adoption of distributed execution platforms like Dryad/DryadLINQ, MapReduce/Sawzall, Hadoop/Pig [18, 29, 7, 25, 12, 24], recent work has investigated job patterns and resource utilization in data centers [1, 14, 22, 23, 26]. These investigation of real work loads have revealed a vast amount of wastage in datacenters due to redundant computations, which is consistent with our findings from logs of a number of production clusters.

DryadInc [26] represented our early attempt to eliminate redundant computations via caching, even before we started on the DryadLINQ project. The caching approach is quite similar to Nectar. However, it works at the level of Dryad dataflow graph, which is too general and too low-level for the system we wanted to build.

The two systems that are most related to Nectar are the stateful bulk processing system described by Logothetis et al. [22] and Comet [14]. These systems mainly focus on addressing the important problem of incremental computation, which is also one of the problems Nectar is designed to address. However, Nectar is a much more ambitious system, attempting to provide a comprehensive solution to the problem of automatic management of data and computation in a datacenter.

As a design principle, Nectar is designed to be transparent to the users. The stateful bulk processing system takes a different approach by introducing new primitives and hence makes *state* explicit in the programming model. It would be interesting to understand the trade-offs in terms of performance and ease of programming.

Comet, also built on top of Dryad and DryadLINQ, also attempted to address the sub-computation problem by co-scheduling multiple programs with common subcomputations to execute together. There are two interesting issues raised by the paper. First, when multiple programs are involved in caching, it is difficult to determine if two code segments from different programs are identical. This is particularly hard in the presence of user-defined functions, which is very common in the kind of DryadLINQ programs targeted by both Comet and Nectar. It is unclear how this determination is made in Comet. Nectar addresses this problem by building a sophisticated static program analyzer that allows us to compute the dependency of user-defined code. Second, co-scheduling in Comet requires submissions of multiple programs with the same timestamp. It is therefore not useful in all scenarios. Nectar instead shares subcomputations across multiple jobs executed at different times by using a datacenter-wide, persistent cache service.

Caching function calls in a functional programming language is well studied in the literature [15, 21, 27]. Memoization avoids re-computing the same function calls by caching the result of past invocations. Caching in Nectar can be viewed as function caching in the context of large-scale distributed computing.

6 Discussion and Conclusions

In this paper, we described Nectar, a system that automates the management of data and computation in datacenters. The system has been deployed on a 240-node research cluster, and has been in use by a small number of developers. Feedback has been quite positive. One very popular comment from our users is that the system makes program debugging much more interactive and fun. Most of us, the Nectar developers, use Nectar to develop Nectar on a daily basis, and found a big increase in our productivity.

To validate the effectiveness of Nectar, we performed a systematic analysis of computation logs from 25 production clusters. As reported in Section 4, we have seen huge potential value in using Nectar to manage the computation and data in a large datacenter. Our next step is to work on transferring Nectar to Microsoft production datacenters.

Nectar is a complex distributed systems with multiple interacting policies. Devising the right policies and fine-tuning their parameters to find the right trade-offs is essential to make the system work in practice. Our evaluation of these tradeoffs has been limited, but we are actively working on this topic. We hope we will continue to learn a great deal with the ongoing deployment of Nectar on our 240-node research cluster.

One aspect of Nectar that we have not explored is that it maintains the provenance of all the derived datasets in the datacenter. Many important questions about data provenance could be answered by querying the Nectar cache service. We plan to investigate this further in future work.

What Nectar essentially does is to unify computation and data, treating them interchangeably by maintaining the dependency between them. This allows us to greatly improve the datacenter management and resource utilization. We believe that it represents a significant step forward in automating datacenter computing.

Acknowledgments

We would like to thank Dennis Fetterly and Maya Haridasan for their help with TidyFS. We would also like to thank Martín Abadi, Surajit Chaudhuri, Yanlei Diao, Michael Isard, Frank McSherry, Vivek Narasayya, Doug Terry, and Fang Yu for many helpful comments. Thanks also to the OSDI review committee and our shepherd Pei Cao for their very useful feedback.

References

[1] AGRAWAL, P., KIFER, D., AND OLSTON, C. Scheduling shared scans of large data files. *Proc. VLDB Endow. 1*, 1 (2008), 958–969.

[2] AGRAWAL, S., CHAUDHURI, S., AND NARASAYYA, V. R. Automated selection of materialized views and indexes in SQL databases. In *VLDB* (2000), pp. 496–505.

[3] ALVARO, P., CONDIE, T., CONWAY, N., ELMELEEGY, K., HELLERSTEIN, J. M., AND SEARS, R. Boom analytics: exploring data-centric, declarative programming for the cloud. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems* (2010), pp. 223–236.

[4] BRODER, A. Z. Some applications of Rabinś fingerprinting method. In *Sequences II: Methods in Communications, Security, and Computer Science* (1993), Springer-Verlag, pp. 143–152.

[5] CERI, S., AND WIDOM, J. Deriving production rules for incremental view maintenance. In *VLDB '91: Proceedings of the 17th International Conference on Very Large Data Bases* (1991), pp. 577–589.

[6] CHAIKEN, R., JENKINS, B., LARSON, P.-A., RAMSEY, B., SHAKIB, D., WEAVER, S., AND ZHOU, J. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow. 1*, 2 (2008), 1265–1276.

[7] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Commun. ACM 51*, 1 (2008), 107–113.

[8] FETTERLY, D., HARIDASAN, M., ISARD, M., AND SUNDARARAMAN, S. TidyFS: A simple and small distributed filesystem. Tech. Rep. MSR-TR-2010-124, Microsoft Research, October 2010.

[9] GOLUMBIC, M. C. *Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathematics, Vol. 57)*. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 2004.

[10] GRAY, J., CHAUDHURI, S., BOSWORTH, A., LAYMAN, A., REICHART, D., VENKATRAO, M., PELLOW, F., AND PIRAHESH, H. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery 1*, 1 (1997).

[11] GRAY, J., SZALAY, A., THAKAR, A., KUNSZT, P., STOUGHTON, C., SLUTZ, D., AND VANDENBERG, J. Data mining the SDSS SkyServer database. In *Distributed Data and Structures 4: Records of the 4th International Meeting* (Paris, France, March 2002), Carleton Scientific, pp. 189–210. Also available as MSR-TR-2002-01.

[12] The Hadoop project. `http://hadoop.apache.org/`.

[13] HALEVY, A. Y. Answering Queries Using Views: A Survey. *VLDB J. 10*, 4 (2001), 270–294.

[14] HE, B., YANG, M., GUO, Z., CHEN, R., SU, B., LIN, W., AND ZHOU, L. Comet: batched stream processing for data intensive distributed computing. In *ACM Symposium on Cloud Computing (SOCC)* (2010), pp. 63–74.

[15] HEYDON, A., LEVIN, R., MANN, T., AND YU, Y. *Software Configuration Management Using Vesta*. Springer-Verlag, 2006.

[16] HEYDON, A., LEVIN, R., AND YU, Y. Caching function calls using precise dependencies. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation* (New York, NY, USA, 2000), ACM, pp. 311–320.

[17] The HIVE project. `http://hadoop.apache.org/hive/`.

[18] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (2007), pp. 59–72.

[19] LEE, K. Y., SON, J. H., AND KIM, M. H. Efficient incremental view maintenance in data warehouses. In *CIKM '01: Proceedings of the tenth international conference on Information and knowledge management* (2001), pp. 349–356.

[20] The LINQ project. `http://msdn.microsoft.com/netframework/future/linq/`.

[21] LIU, Y. A., STOLLER, S. D., AND TEITELBAUM, T. Static caching for incremental computation. *ACM Trans. Program. Lang. Syst. 20*, 3 (1998), 546–585.

[22] LOGOTHETIS, D., OLSTON, C., REED, B., WEBB, K., AND YOCUM, K. Stateful bulk processing for incremental algorithms. In *ACM Symposium on Cloud Computing (SOCC)* (2010).

[23] OLSTON, C., REED, B., SILBERSTEIN, A., AND SRIVASTAVA, U. Automatic optimization of parallel dataflow programs. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference* (2008), pp. 267–273.

[24] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (2008), pp. 1099–1110.

[25] PIKE, R., DORWARD, S., GRIESEMER, R., AND QUINLAN, S. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming 13*, 4 (2005).

[26] POPA, L., BUDIU, M., YU, Y., AND ISARD, M. DryadInc: Reusing work in large-scale computations. In *Workshop on Hot Topics in Cloud Computing (HotCloud)* (San Diego, CA, June 15 2009).

[27] PUGH, W., AND TEITELBAUM, T. Incremental computation via function caching. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1989), pp. 315–328.

[28] YU, Y., GUNDA, P. K., AND ISARD, M. Distributed aggregation for data-parallel computing: interfaces and implementations. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), pp. 247–260.

[29] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, Ú., GUNDA, P. K., AND CURREY, J. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)* (2008), pp. 1–14.